Alexandria University
Faculty of Engineering
Computer and Communications Department

CC423: Advanced Computer Architecture

Sheet 3: Instruction-Level Parallelism and Its Exploitation

---

1. What would be the baseline performance (in cycles, per loop iteration) of the code sequence in Figure 1 if no new instruction's execution could be initiated until the previous instruction's execution had completed? Ignore front-end fetch and decode. Assume for now that execution does not stall for lack of the next instruction, but only one instruction/cycle can be issued. Assume the branch is taken, and that there is a one-cycle branch delay slot.

			Latencies beyond single cycle	
Loop:	LD	F2,0(RX)	Memory LD	+4
I0:	DIVD	F8,F2,F0	Memory SD	+1
I1:	MULTD	F2,F6,F2	Integer ADD, SUB	+0
I2:	LD	F4,0(Ry)	Branches	+1
I3:	ADDD	F4,F0,F4	ADDD	+1
I4:	ADDD	F10,F8,F2	MULTD	+5
I5:	ADDI	Rx,Rx,#8	DIVD	+12
I6:	ADDI	Ry,Ry,#8		
I7:	SD	F4,0(Ry)		
I8:	SUB	R20,R4,Rx		
I9:	BNZ	R20,Loop		

Figure 1

2. Think about what latency numbers really mean—they indicate the number of cycles a given function requires to produce its output, nothing more. If the overall pipeline stalls for the latency cycles of each functional unit, then you are at least guaranteed that any pair of back-to-back instructions (a "producer" followed by a "consumer") will execute correctly. But not all instruction pairs have a producer/consumer relationship. Sometimes two adjacent instructions have nothing to do with each other. How many cycles would the loop body in the code sequence in Figure 1 require if the pipeline detected true data dependences and only stalled on those, rather than blindly stalling everything just because one functional unit is busy? Show the

code with <stall > inserted where necessary to accommodate stated latencies. (Hint: An instruction with latency +2 requires two <stall > cycles to be inserted into the code sequence. Think of it this way: A one-cycle instruction has latency 1 + 0, meaning zero extra wait states. So, latency 1 + 1 implies one stall cycle; latency 1 + N has N extra stall cycles.

3. Consider a multiple-issue design. Suppose you have two execution pipelines, each capable of beginning execution of one instruction per cycle, and enough fetch/decode bandwidth in the front end so that it will not stall your execution. Assume results can be immediately forwarded from one execution unit to another, or to itself. Further assume that the only reason an execution pipeline would stall is to observe a true data dependency. Now how many cycles does the loop require?

4. Reorder the instructions to improve performance of the code in Figure 1. Assume the two-pipe machine in Exercise 3 has been dealt with successfully. Just worry about observing true data dependences and functional unit latencies for now. How many cycles does your reordered code take?

5. Every cycle that does not initiate a new operation in a pipe is a lost opportunity, in the sense that your hardware is not living up to its potential.
    a. In your reordered code from Exercise 4, what fraction of all cycles, counting both pipes, were wasted (did not initiate a new op)?
    b. Loop unrolling is one standard compiler technique for finding more parallelism in code, in order to minimize the lost opportunities for performance. Hand-unroll two iterations of the loop in your reordered code from Exercise 4.
    c. What speedup did you obtain? (For this exercise, just color the N + 1 iteration's instructions green to distinguish them from the Nth iteration's instructions; if you were actually unrolling the loop, you would have to reassign registers to prevent collisions between the iterations.)

6. In this exercise, we will look at the simple register renaming: when the hardware register renamer sees a source register, it substitutes the

destination T register of the last instruction to have targeted that source register. When the rename table sees a destination register, it substitutes the next available T for it, but superscalar designs need to handle multiple instructions per clock cycle at every stage in the machine, including the register renaming. A simple scalar processor would therefore look up both src register mappings for each instruction and allocate a new dest mapping per clock cycle. Superscalar processors must be able to do that as well, but they must also ensure that any dest -to-src relationships between the two concurrent instructions are handled correctly. Consider the sample code sequence in Figure 2 . Assume that we would like to simultaneously rename the first two instructions. Further assume that the next two available T registers to be used are known at the beginning of the clock cycle in which these two instructions are being renamed. Conceptually, what we want is for the first instruction to do its rename table lookups and then update the table per its destination's T register. Then the second instruction would do exactly the same thing, and any interinstruction dependency would thereby be handled correctly. But there's not enough time to write that T register designation into the renaming table and then look it up again for the second instruction, all in the same clock cycle. That register substitution must instead be done live (in parallel with the register rename table update). Figure 3 shows a circuit diagram, using multiplexers and comparators, that will accomplish the necessary on-the-fly register renaming. Your task is to show the cycle-by-cycle state of the rename table for every instruction of the code shown in Figure 2 . Assume the table starts out with every entry equal to its index (T0 = 0 ; T1 = 1 , …).

```
I0:   SUBD    F1,F2,F3
I1:   ADDD    F4,F1,F2
I2:   MULTD   F6,F4,F1
I3:   DIVD    F0,F2,F6
```

Figure 2

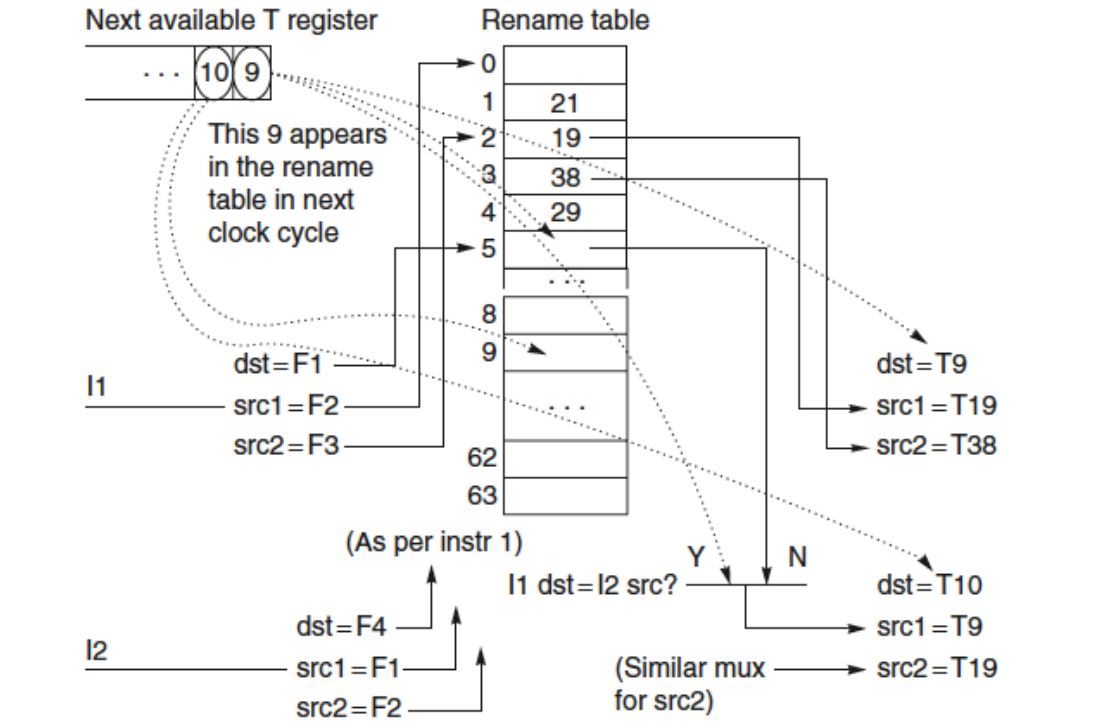

Figure 3

7. Assume a five-stage single-pipeline microarchitecture (fetch, decode, execute, memory, write-back) and the code in Figure 4 . All ops are one cycle except LW and SW , which are 1 + 2 cycles, and branches, which are 1 + 1 cycles. There is no forwarding. Show the phases of each instruction per clock cycle for one iteration of the loop.
   a. How many clock cycles per loop iteration are lost to branch overhead?
   b. Assume a static branch predictor, capable of recognizing a backwards branch in the Decode stage. Now how many clock cycles are wasted on branch overhead?
   c. Assume a dynamic branch predictor. How many cycles are lost on a correct prediction?

```
Loop:  LW      R3,0(R0)
       LW      R1,0(R3)
       ADDI    R1,R1,#1
       SUB     R4,R3,R2
       SW      R1,0(R3)
       BNZ     R4, Loop
```

Figure 4

8. In this exercise, we will look at how variations on Tomasulo's algorithm perform when running the following loop and The functional units (FUs) are described in the table below.

```
        DADDIU   R4,R1,#800   ;  R1 = upper bound for X
foo:    L.D      F2,0(R1)     ; (F2) = X(i)
        MUL.D    F4,F2,F0     ; (F4) = a*X(i)
        L.D      F6,0(R2)     ; (F6) = Y(i)
        ADD.D    F6,F4,F6     ; (F6) = a*X(i) + Y(i)
        S.D      F6,0(R2)     ; Y(i) = a*X(i) + Y(i)
        DADDIU   R1,R1,#8     ; increment X index
        DADDIU   R2,R2,#8     ; increment Y index
        DSLTU    R3,R1,R4     ; test: continue loop?
        BNEZ     R3,foo       ; loop if needed
```

FU Type	Cycles in EX	Number of FUs	Number of reservation stations
Integer	1	1	5
FP adder	10	1	3
FP multiplier	15	1	2

Assume the following:

- Functional units are not pipelined.
- There is no forwarding between functional units; results are communicated by the common data bus (CDB).
- The execution stage (EX) does both the effective address calculation and the memory access for loads and stores. Thus, the pipeline is IF/ID/IS/EX/WB.
- Loads require one clock cycle.
- The issue (IS) and write-back (WB) result stages each require one clock cycle.
- There are five load buffer slots and five store buffer slots.
- Assume that the Branch on Not Equal to Zero (BNEZ) instruction require one clock cycle.

a. For this problem use the single-issue Tomasulo MIPS pipeline of Figure 5 with the pipeline latencies from the table above. Show the number of stall cycles for each instruction and what clock cycle each instruction begins execution (i.e., enters its first EX cycle) for

three iterations of the loop. How many cycles does each loop iteration take? Report your answer in the form of a table with the following column headers:

- Iteration (loop iteration number)
- Instruction
- Issues (cycle when instruction issues)
- Executes (cycle when instruction executes)
- Memory access (cycle when memory is accessed)
- Write CDB (cycle when result is written to the CDB)
- Comment (description of any event on which the instruction is waiting) Show three iterations of the loop in your table. You may ignore the first instruction.

b. Repeat part (a) but this time assume a two-issue Tomasulo algorithm and a fully pipelined floating-point unit (FPU).

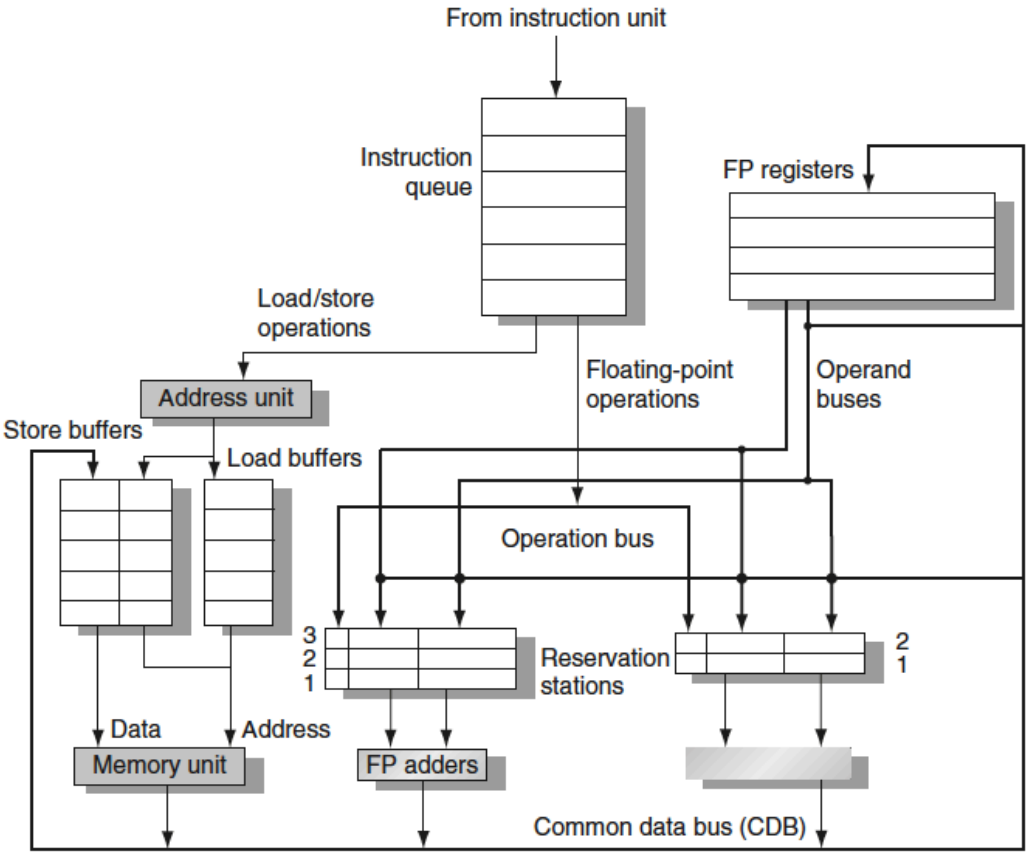

Figure 5

9. Tomasulo's algorithm has a disadvantage: Only one result can compute per clock per CDB. Use the hardware configuration and latencies from the previous question and find a code sequence of no more than 10 instructions where Tomasulo's algorithm must stall due to CDB contention. Indicate where this occurs in your sequence.

10. Suppose we have a deeply pipelined processor, for which we implement a branch-target buffer for the conditional branches only. Assume that the misprediction penalty is always four cycles and the buffer miss penalty is always three cycles. Assume a 90% hit rate, 90% accuracy, and 15% branch frequency. How much faster is the processor with the branch-target buffer versus a processor that has a fixed two-cycle branch penalty? Assume a base clock cycle per instruction (CPI) without branch stalls of one.

11. Consider a branch-target buffer that has penalties of zero, two, and two clock cycles for correct conditional branch prediction, incorrect prediction, and a buffer miss, respectively. Consider a branch-target buffer design that distinguishes conditional and unconditional branches, storing the target address for a conditional branch and the target instruction for an unconditional branch.
    a. What is the penalty in clock cycles when an unconditional branch is found in the buffer?
    b. Determine the improvement from branch folding for unconditional branches. Assume a 90% hit rate, an unconditional branch frequency of 5%, and a two-cycle penalty for a buffer miss. How much improvement is gained by this enhancement? How high must the hit rate be for this enhancement to provide a performance gain?